# GuardAIn: Protecting Emerging Generative AI Workloads on Heterogeneous NPU

Aritra Dhar[1†] Clément Thorens[2†*] Lara Magdalena Lazier[1] Lukas Cavigelli[1]

[1]*Computing System Labs, Huawei Zurich Research Center* [2]*ETH Zurich*

{*aritra.dhar, lara.magdalena.lazier2, lukas.cavigelli*}*@huawei.com* *clement.thorens@inf.ethz.ch*

***Abstract*—Driven by recent advances in large language models (LLMs), generative AI applications have become the dominant workload for the modern cloud. Specialized hardware accelerators, such as GPUs, NPUs, and TPUs, play a key role in AI adoption due to their superior performance over general-purpose CPUs. AI models and the data are often highly sensitive and come from mutually distrusting parties. Existing industry-standard CPU-based TEEs, such as Intel SGX or AMD SEV, do not adequately protect these accelerators. Device-TEEs like Nvidia-CC only address tightly coupled CPU-GPU systems with a proprietary solution requiring TEE on the host CPU side. On the other hand, existing academic proposals target specific CPU-TEE platforms.**

**To address this gap, we propose GuardAIn, a confidential computing architecture for discrete NPU devices that requires no trust in the host system. GuardAIn secures data, model parameters, and operator binaries through authenticated encryption. GuardAIn uses delegation-based memory semantics to ensure isolation from the host software stack, while task attestation guarantees strong model integrity. Our GuardAIn implementation and evaluation with state-of-the-art LLMs such as Llama2 and Llama3 shows that GuardAIn introduces minimal overhead with no changes in the AI software stack.**

## 1. Introduction

Generative AI (GenAI) has gained momentum, with large language models (LLMs) being used in applications, such as chatbots [1], image and video generation [2], [3], and code completion [4]. Major cloud providers offer AI-centric services [5], [6], [7], [8], that typically utilize specialized accelerators such as GPUs, NPUs, and TPUs.

**Security concerns.** GenAI brings numerous security challenges in cloud environments. The massive data and computational resources for training LLMs make them exceedingly expensive [9] and model providers' prime intellectual properties. Additionally, users' queries to LLMs often contain sensitive information such as health data, personal information, or business secrets [10]. In a typical cloud deployment setting, there are three mutually distrusting parties: (1) the *model provider* who develops and owns the AI model, (2) the *data provider* who delivers their data to the model for processing, and (3) the *cloud provider* who owns the computing infrastructure where the model runs. In this context, the model and data need protection from the cloud provider and the corresponding software stack.

**Gap in prior work.** Existing CPU-based trusted execution environments (TEE) [11], [12], [13], [14] isolates secure applications or enclaves from a malicious privileged software stack, such as the OS or the hypervisor. In addition, memory encryption protects untrusted DRAM and the system bus. Besides CPUs, various works have proposed TEEs on devices such as GPUs [15], [16], [17], IPUs [18] or FPGAs [19], [20], [21]. Some [22], [23], [24], [25], [26] also extend CPU-TEE's security primitives to accelerators. However, with a few exceptions [18], [20], existing proposals require a CPU-TEE, which increases the Trusted Computing Base (TCB). Besides increasing TCB, CPU-TEE security guarantees can be undermined by side-channel attacks [27], [28], [29], [30], [31]. Additionally, solutions requiring a confidential VM (C-VM), such as CCA and SEV, further increase the TCB by trusting the C-VM OS. Several proposals focus solely on integrated GPUs [23], [25], [26] or NPUs [24], [32], a more straightforward setting that does not consider PCIe communication or separate memory spaces from the CPU. Placing a part of the driver inside a high-privilege trusted security monitor [24], [25], [33] offloads critical security decisions such as memory allocation for tasks and binaries, memory sharing, and access control to a trusted entity. Such designs expand the TCB and do not fit well in a scenario where the host is fully malicious. While a proposal on Graphcore IPU [18] supports a TEE-less host, it lacks a modern AI software stack with interactive sessions and necessitates extensive hardware modifications. Finally, almost all the existing TEE proposals consider older CNN-based AI models or evaluate using operators such as matrix multiplication or SVD, which have a small memory footprint. Therefore, these solutions do not scale to practical LLMs with large memory footprints and low latency response requirements.

**Our contribution.** We design GuardAIn, a confidential computing solution for discrete NPUs *without* relying on a CPU-TEE. The TCB of GuardAIn consists solely of the NPU itself, while the entire host is untrusted. The hardware root of trust (HW-RoT) in the NPU facilitates attestation and key derivation to establish separate secure channels between the model and the data provider. Measured boot ensures the

†Joint first authors.
*Work done while the author was at Huawei Zurich Research Center.

NPU boots with the correct firmware signed by the hardware manufacturer. GuardAIn accepts fully encrypted data and models from the (mutually-distrusting) data and model provider. During inference, the NPU removes all the DMA mapping from the host's virtual address space (by removing SMMU entries) to prevent malicious DMA operations from accessing the model, data, and workspace (operator execution space). Data and model decryption begin once the memory has been unmapped. The results are encrypted before the host maps back the corresponding memory. The NPU runtime creates tasks from the AI model that specifies the order of operator execution (e.g., ReLU $\rightarrow$ matrix multiplication $\rightarrow$ SoftMax) and memory operation (such as DMA copy from host to device). The malicious host can inject tasks (e.g., performing a DMA copy) into the model to compromise the confidentiality of the data and model. GuardAIn attests to the integrity of the model computation before the execution begins. It provides isolation and end-to-end encryption without introducing changes to the AI software stack, such as PyTorch. Therefore, GuardAIn enables AI programmers to support confidential computing without modifying existing codebases.

We prototype GuardAIn on a Huawei Ascend 910A, a state-of-the-art NPU, by modifying its firmware. We demonstrate GuardAIn on training and inference workloads and evaluate it with state-of-the-art transformer-based LLMs such as GPT-neo-125M, Llama-2-7B (Base and Chat), Llama-3-8B (Base and Instruct), Llama-2-13B-instruct, Llama3-ChatQA-1.5-8B, and CodeLlama-7B-Instruct, for chat, sentence, and code completion. Our evaluation shows that GuardAIn introduces minimal performance loss during the inference pass (0.91% in GPT-neo-125M and 0.028% in Llama3-Chat-QA-1.5-8B model, both with 2K input sequence size) and one-time set-up. Similarly, GuardAIn introduces minimal overhead in training: $3.39\times10^{-4}$% in ResNet152 (CNN) with batch size 256 and 0.16% in NanoGPT-162M (LLM). Although our proposal focuses on a specific NPU implementation, our design philosophy extends to other AI accelerators, such as GPUs and TPUs, that exhibit task-based model execution.

In summary, we make the following contributions:

(1) **Identify design principles for NPU-based confidential computing.** We identify security properties to protect data and models from untrusted host and cloud. Specifically, we design building blocks for confidential computing on an NPU that prevent a privileged host from accessing data and models on the NPU and change their memory mapping during inference and training.

(2) **System design.** Building on these design principles, we develop GuardAIn for discrete NPUs and analyze its security against malicious hosts and cloud providers.

(3) **End-to-end evaluation.** We implement a GuardAIn prototype based on Huawei 910A NPU, evaluate it with state-of-the-art LLMs, and show that GuardAIn introduces minimal overhead in both inference and training with an unmodified AI software stack.

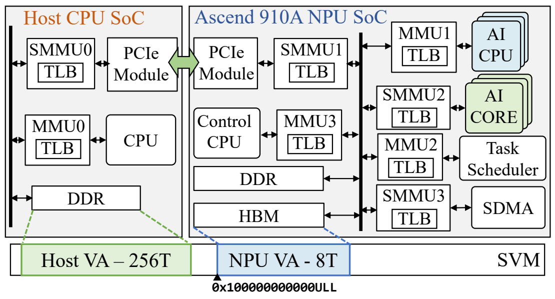


Figure 1. An Ascend 910A SoC's high-level architecture and the shared virtual memory with a 64-bit host CPU.

## 2. Background: AI Accelerators

**AI-accelerators and Task-based Execution.** State-of-the-art high-performance AI accelerators (e.g., NVIDIA [15], [34], GroqChip [35], and Huawei discrete NPUs [36]) have converged on similar heterogeneous system designs. These devices are attached to a host via a bus (for example, PCIe), feature multiple computing units for different operations, and include high-bandwidth memory (HBM). The device drivers handle memory management and communication with the device. Besides the hardware architecture, the software stacks have converged, with the *task-based mode* being established as the de facto standard for AI applications. It breaks a large computation into a series of tasks (commonly referred to as kernels or operators) arranged in a dependency graph. Given a high-level representation of the computation (model), the accelerator's runtime builds and optimizes the graph. It divides the graph into different independent streams, which list interdependent tasks. Each stream has at least one submission queue, allowing a high level of parallelism, as the device can schedule tasks on different queues concurrently. In practice, there are two ways tasks are sent to the device: either one after the other as they become executable (`eager mode`) or all at once (`dynamo` or `graph mode`).

While the low-level implementation and design vary between vendors, the computation paradigm and software architectures described above are not unique to a single vendor. Therefore, while we build our prototype on the Huawei Ascend 910A NPU, the design of GuardAIn can be ported to any accelerator relying on a task-based execution model. The following section will give further details about our specific architecture.

**Ascend 910A.** Huawei Ascend 910 NPU SoC is a state-of-the-art AI SoC for training and inference acceleration on large data centers and clouds. All the components of the SoC are connected via an internal bus, as shown in Fig. 1. The SoC has two types of computation cores that execute AI tasks. Four *AI CPU* cores are general-purpose Huawei Taishan (ARM A73 profile) with hardware cryptographic extensions. Thirty-two *AI Cores* are based on the Huawei DaVinci [37] architecture optimized for executing neural network operations. The control CPU is a Taishan core that runs the NPU firmware and manages the PCIe interface.

The control CPU boots and attests to a minimal Linux kernel using measured boot and initializes all the hardware components. The task scheduler (TS) combines a dedicated hardware component with firmware running on a Taishan core. It distributes the tasks to the AI CPUs and AI Cores.

The NPU runtime sits between the higher-level AI software stack (PyTorch/TensorFlow) and the NPU driver. The NPU driver contains a set of Linux kernel modules, and it manages communication over DMA, issues MMIO commands to send instructions, and monitors NPU health. Ascend PyTorch adapter [38] provides the necessary interface to bridge high-level AI-specific APIs to low-level NPU driver calls. A task contains operator metadata such as the location of the operator binary on the NPU memory (`PC_START`), location of the data arguments, and workspace to store intermediate variables. Using `loadModel` API, the runtime sends the tasks to the task buffer, a reserved NPU memory location. After sending all tasks, the runtime sends an `executeModel` command for `graph mode`, or `compileAndExecute` for `eager mode`. Upon reception of the command, the TS sequentially reads the task buffer, selects the first task, and submits it to the corresponding AI CPU or AI Core. After executing a task, the scheduler moves the task entry to the completion queue (CQ) and continues till the task buffer is empty. The NPU runtime reads the CQ to track the progress of the current execution.

## 3. Motivation, Setting and Attacker Model

**Motivation: Gap in the Prior Work.** Large ML/AI workloads involving sensitive and proprietary data require securing data and computation in the cloud [39], [43]. Notably, the rise of LLMs necessitates confidential computing settings with three parties: data, model provider, and cloud provider. Neither the model nor the data can be leaked to other parties. We call this setting *multi-residence TEE* as the AI workloads running on the TEE access code (i.e., model) and data from different mutually distrusting parties. This is a clear deviation from the traditional cloud-TEE model involving two parties: the enclave user and the untrusted cloud.

Prior works port the confidential computing paradigm to ML-specific accelerators (e.g., NPU [24], GPU [15], [16], [17]). As we show in Table 1, most existing proposals extend security guarantees of CPU-based TEE solutions, such as Intel-SGX [16], [17], AMD-SEV [41], TrustZone [25], [26] or ARM CCA [22], [23], from the host to the device. They have the advantage of including a trusted component on the host to ensure secure communication with the trusted device.

CPU-based TEE significantly enlarges the TCB, requiring trust in the accelerator, the CPU, and respective monitors. Confidential VMs (TDX, CCA) also require trust in the guest OS and driver. Moreover, it reduces compatibility, as the solution relies on a specific CPU on the host, and existing attacks [27], [28], [29] may undermine the CPU-TEEs' security guarantees.

To the best of our knowledge, only two previous works assume untrusted hosts and use the device as the sole hardware root-of-trust: SheF [20] and Graphcore IPU [18]. SheF uses an FPGA as the trusted device, ensuring integrity and confidentiality through authenticated and encrypted bitstreams, along with device isolation. Graphcore relies on a specialized compiler to convert the model into an encrypted and authenticated binary in a clean room environment, which can be sent directly to the device without host intervention. It requires significant hardware changes and prevents interactive AI software stacks. Therefore, these systems are impractical for real-world, large-scale models.

Our proposal *solely* relies on the NPU as the root of trust, *does not* require a CPU-TEE or hardware changes, works with current AI frameworks, extends to other task-based AI accelerators, and is optimized to run modern LLMs.

**A case against spatial sharing.** The increasing scale of LLMs influences resource-sharing and utilization strategies. Earlier ML-specific TEEs focused on spatial sharing (or *multi-tenancy*) based on complex techniques (multiple page tables, monitors, dedicated hardware support) to facilitate resource and performance isolation and boost utilization. We observe this trend in commercial confidential computing solutions such as Nvidia-CC on H100 and B100 (using MIG [44]), as well as several academic proposals [16], [22], [23], [24], [25]. Multi-tenancy is often a choice for workloads with low memory utilization, such as older CNNs (ResNet, VGG, AlexNet, and MobileNet) or isolated operations like matrix multiplication or SVD. However, such workloads do not represent modern AI workloads like LLMs. Table 2 shows the high memory utilization of state-of-the-art LLMs on different GPUs [45] and Huawei Ascend 910A NPU. We observe that most commercially available accelerators have memory ranging from 32 GB to 90 GB, sufficient to run models with 7 to 70 billion parameters (c.f. Table 2). We further observe that a single Ascend 910A NPU has insufficient memory to execute a 13B parameter model with a >2K input sequence length or to load a 70B parameter model. Internal data structures, such as the KV cache, grow quadratically with the input sequence length. LLM applications such as chat-bots [1], code generation [4], or search [46], [47] are latency-sensitive. Lower compute resources due to spatial sharing between multiple tenants result in a higher latency response (e.g., the effective memory bandwidth reduction is proportional to the number of MIG slices [44]). Additionally, supporting a single tenant reduces the hardware and software complexity, as it does not require mechanisms to split NPU resources between tenants. Moreover, single tenancy eliminates the requirement of a TEE (such as a c-VM in case of Nvidia-MIG) as it does not require isolation of tenants within the accelerator. Such a design is also well-suited for our attacker model, where we assume the host is fully attacker-controlled. Given these tradeoffs, we design GuardAIn to be a single-tenant solution. Not supporting multi-tenancy is a limitation of GuardAIn. We discuss more challenges of supporting multi-tenancy in Sec. 6.

TABLE 1. COMPARISON WITH EXISTING CONFIDENTIAL COMPUTING MECHANISMS ON SPECIALIZED ACCELERATORS AND THEIR SECURITY.

| | | | | | |
|---|---|---|---|---|---|
| ●: TRUSTED DRIVER | ◑: PARTIALLY TRUSTED DRIVER | ○: UNTRUSTED DRIVER | ✓: SUPPORTED | ✗: NOT SUPPORTED | ?: UNKNOWN |
| C-VM: CONFIDENTIAL VM | SM: SECURITY MONITOR | HROT: HARDWARE ROOT-OF-TRUST | SB: SECURE BOOT | RA: REMOTE ATTESTATION | LA: LOCAL ATTESTATION |
| STA: SINGLE TASK ATTESTATION | TA: TASK ATTESTATION | SC: SECURITY CONTROLLER | PT: PYTORCH | TF: TENSORFLOW | —: NO CHANGES |

| Existing systems | CC capability and trust assumption | | | | | Device | | AI/ML programming capability | | Required changes | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Host TCB | Isolation granularity | Spatial sharing | Multi-residence TEE | Attestation (CPU/host excl.) | Type | Interface | Native programming interface | AI stack | HW | SW |
| Graviton [16] | Intel SGX + ○ | GPU contexts | ✓ | ✗ | HRoT,RA,STA | GPU | PCIe | CUDA | ? | SC | Runtime, drivers, CUDA |
| HIX [17] | Intel SGX + ● | Enclaves | ✓ | ✗ | LA | GPU | PCIe | CUDA | ? | SGX instruction, MMU, PCIe | GPU enclave, inter-enclave communication, CUDA |
| GraphcoreIPU [18] | ○ | Device | ✗ | ✓ | HRoT,SB,RA | IPU | PCIe | Proprietary | TF | CCU | XLA, poplar compiler, runtime |
| NvidiaCC (H100) [15] | C-VM + ● | VM | ✓ | ✗ | HRoT,SB,RA | GPU | PCIe | CUDA | TF/PT | Security processor | CUDA, C-VM |
| Apple PCC [39] | Enclave + PCC-OS + TXM [40] + ● | Node | ? | ? | HRoT,SB,RA | NPU | Internal | Swift | Proprietary support | Custom Apple Silicon | SepOS, SW stack |
| sNPU [24] | Penglai Enclave + ◑ + SM | Worlds | ✓ | ✗ | HRoT,SB | NPU | Internal | Proprietary | ? | SoC-NoC, SC | SMMU, SM |
| StrongBox [25] | TrustZone + ◑ + SM | Worlds | ✗ | ✗ | SB | GPU | Internal | OpenCL | ? | - | Runtime, driver, MMIO, TASK protector |
| Honeycomb [41] | AMD-SEV + ○ + Validator + SM | SVSM + SM | ✓ | ✗ | SB,RA(of SM) | GPU | PCIe | HIP | ? | - | Validator, SVSM, SM, runtime |
| CAGE [23] | ARM CCA + ◑ + SM | Realm | ✓ | ✗ | RA | GPU | Internal | OpenCL | ? | - | API, monitors, ShadowTask |
| ACAI [22] | ARM CCA + ● + PCIe port | Realm | ✗ | ✗ | SB,RA | GPU + FPGA | PCIe | CUDA | ? | - | TF-A, SMMU, RMM |
| GR-T [26] | Trustzone + ● + cloudVM | VM + Worlds | ✓ | ✗ | SB,RA | GPU | Internal | GlobalPlatform API | ? | - | Driver Shim, GPU_shim |
| HETEE [42] | ● + SM | Node | ✗ | ✗ | SB,RA | GPU | PCIe | CUDA | TF/PT | HETEE box, PCIe interconnect, (SC) | SC, API |
| ShEF [20] | ○ | Device | ✗ | ✗ | HRoT,SB,RA | FPGA | PCIe | ✗ | ✗ | - | ShEF runtime, Shield |
| **GUARDAIN** | ○ | Device | ✗ | ✓ | HRoT,SB,RA,TA | NPU | PCIe | CANN | PT/TF | - | Driver, runtime, kernels (operators) |

TABLE 2. MEMORY UTILIZATION (%) OF LLMS ON DIFFERENT GPUS AND ASCEND 910A NPU WITH #BATCH=1 AND #CONTEXT=512. THE COLOR *GREEN* DENOTES THAT THE MODEL WITH A SPECIFIC QUANTIZATION FITS ON THE ACCELERATOR MEMORY, WHERE COLOR *RED* DENOTES THE MODEL DOES NOT FIT.

| Models | | AI Accelerators | | | | |
|---|---|---|---|---|---|---|
| Variant | Precision size (GB) | 16GB (V100) | 24GB (A5000) | 32GB (**910A**/V100/RTX5090) | 80GB (A100) | 94GB (H100 NVL) |
| llama3-8b | int8 8.7 | 0.5 | 0.4 | 0.3 | 0.1 | 0.1 |
| | fp16 17.4 | 1.1 | 0.7 | 0.5 | 0.2 | 0.2 |
| llama2-13b | int8 14.1 | 0.9 | 0.6 | 0.4 | 0.2 | 0.1 |
| | fp16 28.2 | 1.8 | 1.2 | 0.9 | 0.4 | 0.3 |
| qwen2-14b | int8 15.3 | 1.0 | 0.6 | 0.5 | 0.2 | 0.2 |
| | fp16 30.6 | 1.9 | 1.3 | 1.0 | 0.4 | 0.3 |
| qwen2-32b | int8 33.7 | 2.1 | 1.4 | 1.1 | 0.4 | 0.4 |
| | fp16 67.4 | 4.2 | 2.8 | 2.1 | 0.8 | 0.7 |
| llama3-70b | int8 73.4 | 4.6 | 3.1 | 2.3 | 0.9 | 0.8 |
| | fp16 146.9 | 9.2 | 6.1 | 4.6 | 1.8 | 1.6 |

**Setting, trust assumption, and attacker model.** In a typical trusted execution scenario, the software stack and the cloud provider are untrusted. In addition, our setting involves two types of TEE users: the model and the data provider. The model and data provider are mutually distrusting, and neither trusts the cloud provider:

(**1**) **Cloud/infrastructure provider:** The cloud service provider (CSP) is responsible for provisioning and maintaining all hardware and software resources for operation. The CSP controls all nodes (CPU, NPUs), manages all the infrastructure, including network interfaces, switches, etc., and maintains all the software, such as the OS, hypervisor, device drivers, firmware, and the AI/ML software stack like PyTorch or TensorFlow. We assume all hardware and software the cloud provider provides are *untrusted* except the specific NPUs where the AI model execution occurs.

(**2**) **Model provider:** The model provider develops and trains the model from the ground up (known as the foundation model) and keeps the model's composition and parameters secret. The model provider may also re-train or fine-tune an open-source model with proprietary data to suit new application scenarios (e.g., code generation). Fine-tuning or retraining with secret datasets makes it crucial to keep the model parameters confidential. The CSP is a model provider in the machine learning as a service scenario.

(**3**) **Data owner:** The data owner executes models on cloud infrastructure (CPU, memory, NPU) for training or inference. The models may either be the intellectual property of a model provider or provided by the data owner. We assume that the data and the model provider are mutually distrusting parties.

We only assume NPUs where the deployed AI/ML workloads are *trusted*. The NPUs have an on-chip hardware security module (HSM) that acts as the hardware root of trust. Lastly, we assume that denial of service (DoS) and side-channel attacks are outside the scope of this paper.

## 4. Security Challenges and Requirements

We use the matrix multiplication depicted in Fig. 2 as a running example. The NPU runtime copies the NPU-optimized binary containing a matrix multiplication operator (`torch.mm`), as well as the tensors `M1` and `M2` onto the NPU's memory. After executing the operator, the NPU copies the matrix `M3` from its memory to the CPU's main memory. The three tasks corresponding to this example are represented in Fig. 3, namely, a memory copy of the tensors (`M1`, `M2`) from the host to the NPU, the matrix multiplication and memory copy of the resulting tensor (`M3`)

```
def mm_npu_operator(m, n): #operator definition
    M1 = torch.rand(m,n).npu() #copy M1 to NPU
    M2 = torch.rand(m,n).npu() #copy M2 to NPU
    M3 = torch.mm(M1,M2).cpu() #copy result M3 to CPU
```

Figure 2. An example PyTorch matrix multiplication code.

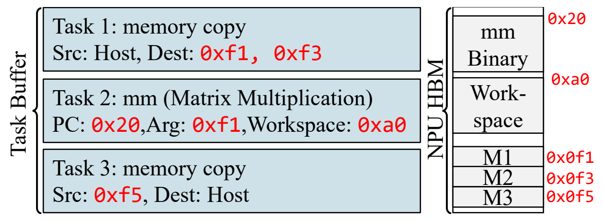


Figure 3. Matrix multiplication task and memory layout on a NPU, corresponding to the code snippet in Fig. 2.

from the NPU to the host memory. The host-side NPU runtime reserves memory spaces on the NPU HBM for the binary of the matrix multiplication operator, tensors (M1, M2, M3), and the workspace (operator's working space, e.g., heap and stack). Based on this execution model, we observe several security challenges assuming a malicious host. We list security challenges and their corresponding requirements for enabling confidential computing on NPUs.

**Security Challenge 1:** The untrusted host runs the privileged softwares (e.g. OS, hypervisor, and device driver) along with an AI software stack. At any point, the untrusted host has full access to the data and model.

→ *Requirement 1*: *End-to-End Authenticated Encryption:* To ensure the attacker-controlled host cannot observe or manipulate the data and model, all inputs, outputs, model parameters, and binaries must be end-to-end encrypted and authenticated. Therefore, the host only handles encrypted data, models, and results at all times.

**Security Challenge 2:** The NPU requires the plaintext model (and data) for execution. When the NPU receives the encrypted model and data, it must first decrypt them. Once the decryption is complete, nothing prevents the host (with full memory access) from reading or modifying them. Thus, we need to ensure the host loses access before the decryption begins. Likewise, end-to-end authenticated encryption is insufficient for the result, as the host can access it before it is encrypted.

→ *Requirement 2*: *Atomic execution invariant:* Before the NPU starts decrypting the data and model, as required prior to execution, the host must lose access to them. This can be achieved by removing the DMA mappings of the NPU memory region where the data and model reside.

**Security Challenge 3:** The host defines the memory mapping for the model's inputs and outputs. If a malicious host declares the output memory region to coincide with the memory region where the model is stored, the data provider gains access to information about the model, compromising its confidentiality.

→ *Requirement 3*: *Memory invariant:* The NPU must reject memory allocations or remapping requests from the host to any memory location that has already been allocated (e.g., model, data, or intermediate results).

**Security Challenge 4:** Even without direct access to the data or model, the untrusted host can issue malicious commands to the NPU, such as copying part of the model parameters into the results accessible by the data provider, compromising the model's confidentiality.

→ *Requirement 4*: *Model attestation:* A lack of integrity in the model execution compromises the model and data security. In particular, one needs to ensure that the host does not change or add any commands or operators.

**Security Challenge 5:** The security primitives for confidential computing are only valid if there is a mechanism to attest the NPU firmware. Without a systematic way to check the integrity of the NPU firmware, we cannot assert the trustworthiness of the NPU's confidential computing capability.

→ *Requirement 5*: NPU attestation: We need a measured boot-equivalent primitive for the NPU, where the NPU only accepts manufacturer-certified firmware and does not allow an attacker-controlled host to flush its firmware or change the runtime configuration.

**Security Challenge 6:** The NPU and its software stack provide several debugging methods for correctness and performance, such as inspecting the NPU memory or monitoring execution time. These mechanisms allow the attacker to extract information about models and data.

→ *Requirement 6*: *Restricted debugging:* To ensure data and model confidentiality, all debugging-related operations must be restricted.

## 5. Basic Building Blocks for Confidential AI

### 5.1. Model and Data Encryption

The model and its associated data must be encrypted with the keys shared between the NPU, the model provider, and the data provider to protect them from an untrusted cloud (Req. 1). The model and data provider are mutually distrusting; they cannot access each other's shared key.

**Setup.** The model and data providers initiate an authenticated Diffie-Hellman key exchange with the NPU. If the model and data provider are the same party, only one key is derived to encrypt the data and model. The NPU derives

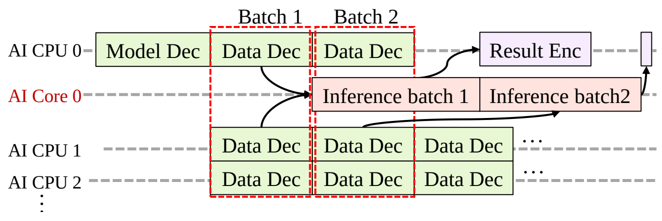


Figure 4. Parallel AES-GCM on the model and data to hide the operator latency running on the AI-CPU cores. The AI core executes the AI-related operators, e.g., matrix multiplication.

ephemeral keys from its root key stored in the hardware RoT, along with the firmware measurement derived in the measured boot (refer to Sec. 5.4). The firmware measurement ensures that the model and data provider interact with a legitimate NPU device running the correct firmware image signed by the hardware vendor.

**5.1.1. AI CPU-based custom operator.** We use the ARM AES intrinsic and the SIMD (NEON) instructions of NPU AI-CPU cores (refer to Sec. 2) to accelerate AES-GCM operations. AES-GCM is implemented as an AI CPU operator executed before (for data and model decryption) and after (for result encryption) the model execution. The AES-GCM operator is part of the NPU firmware, verified by the measured boot during the NPU initialization, and part of the software TCB. Integrating the AES-GCM operator within the NPU firmware ensures that all models utilize the verified encryption implementation, thereby eliminating the need for specialized encryption hardware or modifications to the AI software stack. All the cryptographic operations are performed *in-place* and do not require additional memory on the NPU.

The parallelized AES-GCM operator verifies and decrypts multiple batches of data on multiple AI CPU cores to hide the AES-GCM latency. Fig. 4 shows parallel AES-GCM operations on the model, data, and results running on the AI CPUs, while the AI Cores execute the model layer operators during inference passes. Typically, for an LLM, the computation is bound by the computation layer (a few milliseconds) compared to the AES-GCM operations on the data (in the order of $100\ \mu$ seconds). Therefore, the latency is only visible for the first inference, when the model and the first data batch must be decrypted. The decrypted model is already on the NPU for subsequent inference passes.

**5.1.2. Executing the modified model.** Execution of the modified model takes place in two steps: preparing the model by instrumenting the AI CPU operators, and the actual inference pass of the instrumented model on the NPU.
**Preparing the model.** The compiled model file (or frozen model for inference) contains the layer information, parameters, and operator binaries. The model provider encrypts the weights (a list of `Tensor`) and the individual operator binaries with the secret key shared between the model provider and the NPU. Each binary contains a header, a symbol table, and compiled instructions for the NPU AI core. A list of binary sizes helps the NPU runtime to parse the model file. We modify the binary sizes to account for the 16 bytes of the message authentication code (MAC), as well as the encryption padding.

The model file contains the layer information corresponding to the tasks. For example, the layer named `te_relu_1_1` denotes the first ReLU activation function. A task name typically corresponds to an actual operation, but this is not necessary for the model to function correctly, and the model provider can replace the task names with randomized strings to prevent the underlying operator from being exposed. Note that in our design, the structure of the model will be *visible* to the attacker. This limitation is due to the architecture of the AI runtime, which depends on the model layout to create the computation graph. When the untrusted host receives the encrypted model and data, it copies them to the NPU via the AI software stack (e.g., PyTorch). Following this copy, the host invokes the `executeModel` API to initiate the inference.
**Model execution on the NPU.** The `executeModel` API call from the NPU runtime signals the NPU task scheduler (TS) to start executing the AI model tasks. The NPU TS ensures that the model and data decryption starts right after the call to the `executeModel` API. After one inference pass, i.e., a forward pass throughout all the layers in the model, the model outputs a vector known as the logits. A normalization operation (e.g. `softmax`) on the logits produces the probability values of the inference classes. The AES-GCM operator encrypts the logits before copying them back to the host.

## 5.2. Enforcing Memory Lock Invariants

Decrypting the data and model (refer to Sec. 5.1) right after the `executeModel` call from the untrusted host grants it full access to the sensitive, decrypted content. Therefore, we must revoke the host's access to these NPU memory regions. Requirements Req. 2 and Req. 3 focus on guaranteeing critical memory invariants to ensure the data, model, and execution remain isolated from the attacker-controlled host. We design a memory access control primitive using the NPU's SMMU (similar to the ARM's SMMU [48]). All PCIe transfers between the host and the NPU go through the NPU's SMMU. We enforce access control by modifying the NPU memory manager, located on the control CPU, and with exclusive access to the NPU's SMMU. We need to solve four challenges to ensure secure access control.

**[C1]** We must ensure the following sequence of events: loading the model into NPU memory $\rightarrow$ locking the NPU memory region where the model resides $\rightarrow$ decrypting the model atomically to prevent the host from interrupting the NPU and changing the sequence of events.

**[C2]** Following the decryption of the model and data, their virtual memory spaces can only be unlocked by either re-encrypting or resetting the memory content.

**[C3]** The host cannot map the output of the model to the same memory address space as the input and leak the model parameters and binaries once the output is unlocked.

**[C4]** The host cannot remap the IO memory regions on the NPU to trigger the remapping of the DMA pages on the host after the decryption of the model and data is complete.

For **[C1]**, we ensure that the AES-GCM AI-CPU operator responsible for decrypting the model and data is scheduled only after the NPU memory manager unmaps the model, data, and workspace memory. Unmapping the memory using the `dma_unmap_pages` API call removes all the memory mapping from the NPU's SMMU and blocks any memory access from the host. At this point, any interrupts coming from the host prevent the NPU memory manager from sending an acknowledgment signal to the NPU TS. Under normal circumstances, after receiving the signal, the NPU TS schedules the AES-GCM operator to decrypt and verify the model weights, binaries, and data.

To address **[C2]**, the NPU TS schedules an AES-GCM encryption operator on the outgoing memory location (model output) after the model's execution. Upon completing the encryption, the operator signals the NPU driver to remap the memory using `dma_remap(location, size)`. The remap API adds an entry to the SMMU, allowing the host to copy the encrypted results.

We address **[C3]** by introducing a memory exclusivity invariant within the NPU memory manager. The NPU memory manager contains a data structure that tracks all the allocated memory locations based on their DMA direction (`DMA_BIDIRECTIONAL`, `DMA_TO_DEVICE`, or `DMA_FROM_DEVICE`). By tracking the memory addresses and their corresponding DMA directions, the NPU memory manager prevents the host from declaring an output tensor that points to an already allocated address, i.e., remapping the model address space to the output.

To address **[C4]**, we ensure that all the critical functions, `dma_map`, `dma_unmap`, and `dma_remap`, which can manipulate the NPU's SMMU entries, are *only* callable from the NPU TS. The host communicates with the NPU TS through the model tasks submitted on the task queue. Only the `executeModel` task can trigger the unmapping and subsequent decryption operations. The end of model execution triggers the result encryption and its remapping to the host. This design ensures that the host cannot arbitrarily remap the NPU memory to itself.

## 5.3. Model and Task Attestation

Requirement Req. 4 implies that the integrity of the model execution is critical to protect the integrity and confidentiality of both the model and data. The untrusted NPU driver sends the model (containing layer information, model parameters, and operator binaries) to the NPU memory and writes the tasks on the task queue. The task's `PC_START` attribute points to the operator binary's memory location. As the NPU driver runs on the attacker-controlled host, the host can always push arbitrary tasks and remove or reorder tasks to compromise the integrity of the AI model. The attacker can also modify the tasks' `PC_START` attribute to point to a different binary and thus execute a different operator, compromising the integrity of the model execution. We design a model verification method shown in Fig. 5 to prevent such an attack. We assume two keys $\mathcal{K}_M$ and $\mathcal{K}_D$, shared respectively between the model provider & the NPU, and the data provider & the NPU. We use the matrix multiplication example depicted in Fig. 2, and Fig. 3 to describe our mechanism. Here, the model consists of three layers that execute three operators: a memory copy from host to device (`DMA_TO_DEVICE`), a matrix multiplication, and another memory copy from device to host (`DMA_FROM_DEVICE`). The sequence of the corresponding operators' binaries is $B_1 \rightarrow B_2 \rightarrow B_3$. Fig. 5 depicts the flow of our model and task attestation mechanism. Additionally, a sequence diagram of the same protocol is provided in Fig. 6. The steps are the following:

① An AI model ($M$) consists of layer information ($L$), model parameters ($W$), and operator binaries ($B = \{B_1, B_2, B_3\}$). The model provider encrypts and creates message authentication codes (MAC) of $W$, and of each operator binaries in $B$ such as $\mathcal{B}_i \leftarrow MAC_{\mathcal{K}_M}(B_i)$, and generates the MAC of the concatenated sequence of binary MACs corresponding to the layers, such as $\mathbb{M} \leftarrow Enc_{\mathcal{K}_M}(M)$ and $\mathcal{B} \leftarrow MAC_{\mathcal{K}_M}(\mathcal{B}_1 \| \mathcal{B}_2 \| \mathcal{B}_3)$. $L$ contains the name of the layer (that the model provider can masquerade) and the location of the encrypted binary relative to a fixed starting point in the $M$. This information enables the NPU runtime to generate the corresponding layer tasks. The model provider sends $\mathbb{M}$ and $\mathcal{B}$ to the untrusted host (on the CSP) and $\mathcal{B}$ to the data provider.

② Using `loadModel` API, the host deploys the model to the NPU. The runtime determines `PC`, the starting addresses of the operator binaries $(B_1, B_2, B_3)$ to map them on the NPU memory. In our example, `PC`=$\{$`0x10,0x20,0x30`$\}$.

③ The NPU driver copies the model ($L, W, B_1, B_2, B_3$ in Fig. 5) to the NPU's HBM over DMA. $W$ and $\{B_1, B_2, B_3\}$ are encrypted with $\mathcal{K}_M$. The NPU runtime creates a sequence of tasks: $T_1, T_2, T_3$ from the layer information and uses `PC` to populate the `PC_START` attribute.

④ The untrusted host sends the `PC` to the data provider.

⑤ The data provider generates $P_1 \leftarrow MAC_{\mathcal{K}_D}(\texttt{PC})$ and $P_2 \leftarrow MAC_{\mathcal{K}_D}(\mathcal{B})$ ,and sends $P_1$ and $P_2$ to the host.

⑥ The driver invokes the `executeModel` API by writing a specific execution task ($E$ in Fig. 5) on the task queue. $P_1$ is sent together with `executeModel`. This triggers the task scheduler (TS) to remove all the memory mappings from the NPU's SMMU. Therefore, the host can no longer read from or write to either the HBM or the task queue.

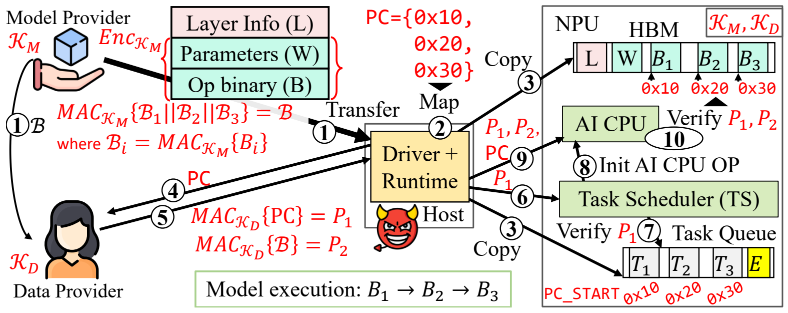
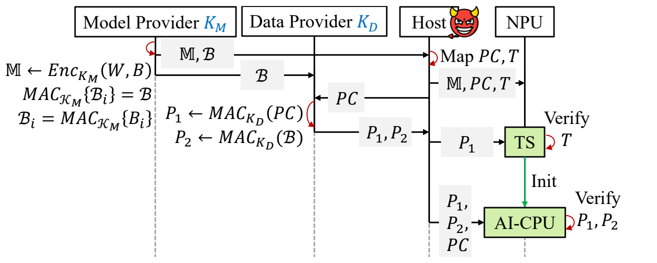


Figure 5. Protocol to ensure the integrity and the confidentiality of the AI model, as well as the integrity of model execution.



Model Provider $K_M$
Data Provider $K_D$
Host
NPU
$\mathbb{M} \leftarrow Enc_{\mathcal{K}_M}(W, B)$
$MAC_{\mathcal{K}_M}\{\mathcal{B}_i\} = \mathcal{B}$
$\mathcal{B}_i = MAC_{\mathcal{K}_M}\{B_i\}$
$\mathbb{M}, \mathcal{B}$
$\mathcal{B}$
Map $PC, T$
$\mathbb{M}, PC, T$
$PC$
$P_1 \leftarrow MAC_{\mathcal{K}_D}(PC)$
$P_2 \leftarrow MAC_{\mathcal{K}_D}(\mathcal{B})$
$P_1, P_2$
$P_1$
TS
Verify $T$
Init
Verify $P_1, P_2$
$P_1, P_2, PC$
AI-CPU

Figure 6. Sequence diagram of GUARDAIN model and task attestation (Fig. 5) to ensure the integrity and the confidentiality of the AI model, as well as the integrity of the model execution.

⑦ The NPU TS collects all the `PC_START` attributes from the task queue. The TS calculates $P_1' \leftarrow MAC_{\mathcal{K}_D}(\texttt{PC\_START})$ and aborts if $P_1 \neq P_1'$.

⑧ If step ⑦ is successful, the TS invokes an AES-GCM AI CPU operator (c.f., Sec. 5.1) to verify the integrity of the sequence of binaries using their MACs ($\mathcal{B}_i$'s).

⑨ The host sends $P_1, P_2$, and `PC` to the AI CPU operator.

⑩ The AI CPU operator checks the correctness of $P_1$ from the NPU TS (verified in step ⑦). Using `PC`, the AI CPU decrypts the binaries individually, checking that they match their associated MAC. Once the binaries are all decrypted, the AI CPU computes $\mathcal{B}' \leftarrow MAC_{\mathcal{K}_M}(\mathcal{B}_1\|\mathcal{B}_2\|\mathcal{B}_3)$ and $P_2' \leftarrow MAC_{\mathcal{K}_D}(\mathcal{B}')$ and checks that $P_2' = P_2$. If these steps are successful, the operator decrypts W in place on the HBM. Removing all memory mappings in the NPU's SMMU in step ⑥ prohibits the host from accessing the decrypted $W$ and $B$.

### 5.4. Firmware and Runtime Integrity

All TEE mechanisms are implemented in the NPU firmware, which is part of GUARDAIN software TCB. Therefore, the trustworthiness of the GUARDAIN depends on the integrity and authenticity of the firmware. The NPU vendor programs a cryptographic key (e-fuse) during the manufacturing process. This cryptographic key functions as the hardware root-of-trust and is *non-extractable*. It serves as an unforgeable identity, preventing the attacker from impersonating and emulating a legitimate NPU. Subsequent keys for key exchanges are derived from this root key. The control CPU initiates the NPU sub-modules during start-up and verifies whether the firmware image (and version) is signed with the manufacturer's root key. This prevents the attacker from flashing an unsigned firmware image to the NPU. We assume the cloud service provider has a public key infrastructure to ensure that the model and data provider can execute an authenticated Diffie-Hellman key exchange with the NPU to derive shared secrets. The shared secret is then used to encrypt and authenticate the model and data, and to verify the legitimacy of the NPU. The NPU control CPU intercepts all the command messages coming from the host runtime. GUARDAIN blocks all debugging commands (e.g., memory inspection, profiling operators) from the control CPU to ensure the attacker has no unmediated communication channel with the NPU.

## 6. GUARDAIN

Based on the building blocks discussed in Sec. 5, we now describe the GUARDAIN end-to-end system.

**Initial setup.** The NPU has an on-chip hardware security module (HSM) that securely stores cryptographic keys and executes operations such as key derivation functions (KDF), shared key derivations (using Diffie-Hellman), digital signature verifications, etc. The HSM contains the NPU's root key in the e-Fuse, which acts as the primary root of trust. Using a KDF, the NPU HSM derives ephemeral keys for every new session. The model and the data provider execute an authenticated Diffie-Hellman key exchange with the NPU (using the derived ephemeral session keys) over the untrusted host and obtain $\mathcal{K}_M$ and $\mathcal{K}_D$. The NPU stores these two keys on the HSM on-chip. During the key exchange, the NPU sends the signed firmware version for attestation. We assume the data provider interacts with the remote LLM application (e.g., chatbot) through a browser loaded with $\mathcal{K}_D$. The browser tokenizes the data provider's input ($D$), encrypts it ($\mathbb{D} \leftarrow Enc_{\mathcal{K}_D}(D)$), and sends the encrypted tokens to the cloud provider. Similarly, the model provider encrypts the model binaries and parameters: $\mathbb{M} \leftarrow Enc_{\mathcal{K}_M}(M)$, and generates the signed sequence of operator binaries for the model and task attestation (Sec. 5.3).

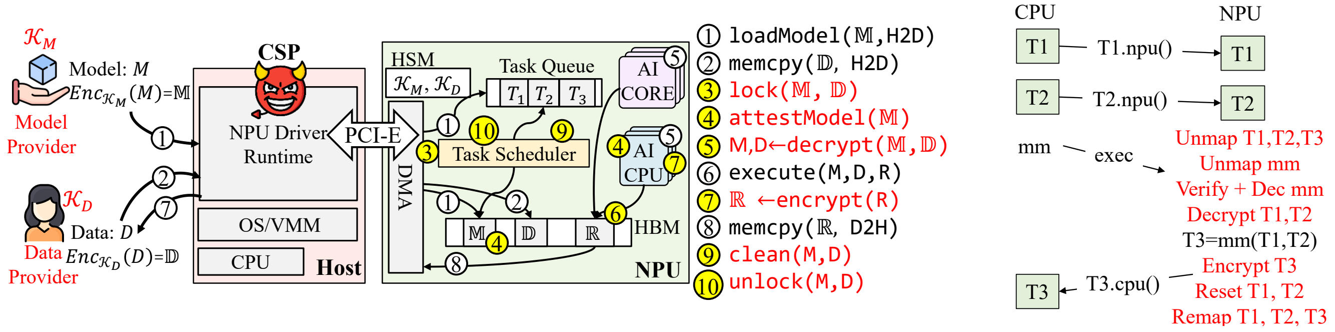


Figure 7. End-to-end GUARDAIN system with internal confidential computing components and the corresponding PyTorch interfaces.

TABLE 3. SECURITY CHALLENGES AND CORRESPONDING GUARDAIN MECHANISMS TO ADDRESS THEM.

| Security challenges (Sec. 4) | GUARDAIN mechanisms (Sec. 5) |
|---|---|
| Data + model confidentiality (SC 1) | AI-CPU crypto operator (Sec. 5.1) |
| Memory Isolation (SC 2 and 3) | Memory locking (Sec. 5.2) |
| Host interrupt (SC 2) | Atomic execution (Sec. 5.2) |
| Model + task integrity(SC 4) | Task and model attestation (Sec. 5.3) |
| Firmware integrity + debug (SC 5 and 6) | Measured boot + disable debug (Sec. 5.4) |

**GUARDAIN: End-to-end system.** Table 3 summarizes security challenges and corresponding mitigation techniques discussed in Sec. 5. We build the GUARDAIN end-to-end system upon these building blocks. Fig. 7 shows the end-to-end GUARDAIN system along with its mechanisms (step ① to step ⑩) and their execution locations. The security-sensitive steps that GUARDAIN adds to the NPU firmware to enable confidential computing are highlighted (●). A description of these steps is as follows: ① the host calls `loadModel` to send the encrypted model $\mathbb{M}$ to the NPU. ② Then `memcpy` transfers the encrypted data $\mathbb{D}$. The host instructs the NPU to execute the AI model with the `executeModel` API. However, as the model and data are encrypted, the execution of the model is not possible without first decrypting both. In GUARDAIN, `executeModel` triggers the following steps to ensure the attacker-controlled host does not manipulate the model tasks or access the model and data after the decryption. ③ `lock`($\mathbb{M}, \mathbb{D}$): The NPU task scheduler (TS) intercepts `executeModel` from the host and instructs the NPU memory manager to remove the model, workspace, and data regions, located on the NPU HBM, from the SMMU mappings. Therefore, the model, data, and workspace on the NPU are no longer accessible from the host. This mechanism is described in Sec. 5.2. ④ `attestModel` ($\mathbb{M}$): The NPU TS verifies the tasks and model binaries. The verification is a multi-step protocol that involves a dedicated AI CPU operator. We describe the model and task attestation mechanism in Sec. 5.3. ⑤ Once the memory regions are locked, the TS invokes an AI CPU operator to decrypt the model and data (refer to Sec. 5.1) in place. ⑥ The AI model is executed on the AI cores and AI CPUs based on the operators in the model layers. The output of the model is $R$. The memory region(s) where $R$ resides are not DMA-mapped to the host. ⑦ The NPU TS invokes the AI CPU cryptographic operator (Sec. 5.1)

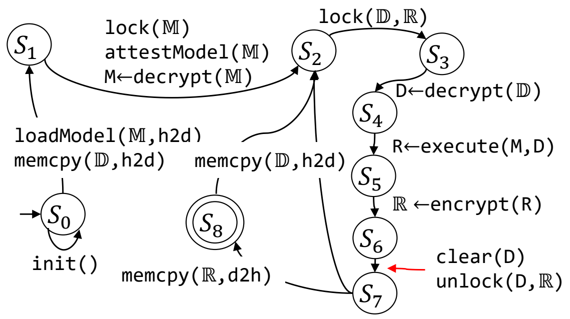


Figure 8. GUARDAIN memory and execution lifecycle.

to encrypt the model output with the data provider's key: $\mathbb{R} \leftarrow Enc_{\mathcal{K}_D}(R)$) and unlocks the region. ⑧ The host issues a `memcpy` command to copy the encrypted output $\mathbb{R}$ from the NPU memory to the host memory. ⑨ If the host runs another inference pass, the TS invokes another AI-CPU operator to zero out the input data. If the host triggers the end of the session, the TS cleans up both the model and input data. ⑩ The TS invokes the NPU memory manager to remap the memory (either $D$ or both $D$ and $M$) before the start of the next inference pass or the end of the session.
**Programming interface.** GUARDAIN changes in the NPU firmware and driver are transparent to the higher-level software stack (e.g., PyTorch). Therefore, from the AI developer's perspective, the existing inference or training workflow remains unchanged. There are minimal changes within the Ascend PyTorch adapter to support the model attestations. Fig. 7 shows the matrix multiplication example (from Fig. 2) with the GUARDAIN mechanisms.
**GUARDAIN lifecycle.** Fig. 8 shows GUARDAIN's memory lifecycle over subsequent inference rounds. Typically, the model is loaded once, followed by multiple inference and training rounds (e.g., queries to an LLM chatbot). Therefore, the memory associated with the model (parameters, binaries, and model operator workspace) must be locked and decrypted only once and remain locked until the model is unloaded or the NPU is reset. On the other hand, the input data arrives in a streaming fashion and needs to be locked, decrypted, and fed to the model in each round. Once the NPU generates the output, it encrypts it with the data owner's key and unlocks the corresponding memory region

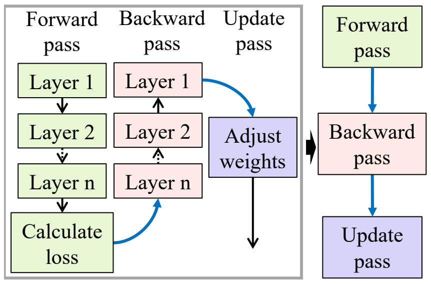


Figure 9. GUARDAIN merges three subgraphs, forward, backward, and update, to a single graph by statically comping all the layer operators.

for the DMA transfer. At the same time, the NPU resets the input region (by overwriting it to zero) and unlocks it so the host can transfer the next batch of encrypted data.

**Scale-out Inference.** We describe GUARDAIN mechanisms for a single NPU inference. However, scaling GUARDAIN to multiple NPUs is feasible. In data-parallel, GUARDAIN runs independently on different NPUs, ingesting data in parallel. For parallel pipeline inference, we need to encrypt the output of one layer with an AI CPU encryption operator and decrypt it on the next NPU. For tensor-parallel, the outcome of all NPUs must be encrypted before broadcasting and decrypted before all-reduce.

**GUARDAIN adaptation for training.** We previously described how GUARDAIN protects the model and data during inference. These basic building blocks remain the same for model training. Training has additional steps compared to inference. Fig. 9 shows an example training pass on an AI model with $n$ layers. At the start of the training, the AI model had uninitialized parameters. The training principle involves adjusting these parameters to minimize classification error during inference, which unfolds in three distinct phases. The *forward* pass is a standard inference pass that evaluates the current classification error of the model. The *backward* pass evaluates the parameters' gradient to reduce the classification error. Finally, the *update* pass updates the parameters based on the gradients calculated in the *backward* pass. Therefore, unlike inference where GUARDAIN verifies (refer to Sec. 5.3) one graph, during training GUARDAIN verifies the task integrity of three subgraphs (forward, backward, and update). The model provider statically compiles the AI model graph ahead of time using the `torch-dag` and `torch.compile` API that generates a compiled directed acyclic graph (DAG) combining all three sub-graphs as shown on Fig. 9.

**Generalization to other AI Accelerators.** Many of the GUARDAIN's design principles apply to other existing AI accelerators. GUARDAIN leverages the heterogeneous architecture of Ascend NPUs to offload cryptographic operators to CPU cores and execute them as part of the AI models' computation graph. Commercial AI accelerators such as Nvidia Grace-Hopper [34], Apple neural engine [49], or AMD Accelerated Processing Units [50] use a tightly-coupled CPU-accelerator heterogeneous design where decoupling cryptographic operators from the AI computation is possible. The computation graph created in PyTorch [51] ahead of time enables inserting the security-critical operators required by GUARDAIN design into existing AI models. Similarly, the task attestation mechanism applies to most existing accelerators [35], [52], [53] as they use a task/command-based architecture. Other FPGA-based AI frameworks [54], [55], [56], [57] could use GUARDAIN design principles due to their similar software stack; however, this claim would necessitate further investigation to be validated.

**GUARDAIN limitations.** GUARDAIN does not allow multi-tenants, i.e., multiple models or single model-multiple inference users on the same NPU. Multi-tenant support could help run multiple small LLMs (0.5-3B parameter size). However, allowing multiple inference users on the same model is not secure, as a malicious model provider can collude with a malicious inference user and copy another user's data to the malicious user's output address space. Such an attack can be mitigated by tagging the NPU memory pages to enforce user ownership and by adding a dynamic taint-tracking mechanism to determine data movement between users. However, by design, the NPU runtime running on the host CPU initiates and manages the NPU page table. This is due to the shared virtual memory design where the NPU memory is abstracted as a part of the host virtual address space (c.f. Fig. 1). Therefore, supporting multi-tenants requires moving this functionality from the host to the NPU, as GUARDAIN assumes the host to be attacker-controlled, which involves significant modifications of the NPU runtime. We note that not supporting multi-tenants is a *limitation* of GUARDAIN and could be addressed by a future extension of GUARDAIN.

As discussed previously, GUARDAIN focuses on workloads running on a single NPU. Except for data-parallel, we note that enabling all other scaling out mechanisms, such as pipeline and tensor parallel in GUARDAIN, requires implementing significant design changes and addressing challenges. One challenge is to encrypt massive amounts [1] of data between NPUs before the all-reduce operation merges the computations. This requires dedicated encryption hardware to support low-latency and high-bandwidth AES-GCM. Therefore, addressing such system challenges is non-trivial and is out of the scope of this paper. We consider the lack of scale-out another *limitation* of GUARDAIN.

## 7. Security Analysis

In this section, we provide an informal security analysis of GUARDAIN and show how it ensures the security of AI models and data from an untrusted host and cloud provider.

**Malicious host and cloud provider.** The host runs the operating system/hypervisor, the NPU driver, and the AI software stack and has full access to the NPU. The host allocates and copies the data, model, and operator binaries to the NPU. Once the host calls the `execute()` API, the

1. The interconnect (HCCS) bandwidth between Ascend 910A NPUs is 480 Gbit/s [58].

NPU task scheduler unmaps the NPU memory. Therefore, the host cannot access the encrypted model and data anymore. The host can interrupt the NPU between the memory locking phase and the model and data decryption phase to disrupt the DMA region's locking. However, before the NPU task scheduler schedules the AI CPU operator to decrypt the model and data, it expects a confirmation from the NPU's memory manager that the memory is inaccessible from the host. Similarly, the host can interrupt the NPU between the result encryption and the memory unlocking to prevent the result encryption from occurring. However, the NPU task scheduler only requests the NPU memory manager to unlock the DMA memory after successfully executing all the AI CPU operations, including the encryption of the results. The task scheduler obtains this confirmation from its completion queue (CQ), which indicates if an operator has succeeded or failed. This atomic property prevents the host from accessing the plaintext model and data on the NPU memory. The host cannot issue a malicious DMA to modify the model and data as they contain a MAC from the model and data provider, nor can it manipulate the tasks after a call to `executeModel` as the memory is locked. The NPU control CPU disables all the debugging and performance monitoring interfaces to prevent the host from having additional communication channels with the NPU. Since all data entering and leaving the NPU is encrypted and authenticated, a malicious cloud provider with physical access cannot compromise the device's security. The host cannot manipulate the tasks to change the model execution or leak the model and data, as our task attestation mechanism (Sec. 5.3) prevents such attacks. A malicious cloud provider can flash an NPU card with an older or compromised firmware. However, as described in Sec. 5.4, the NPU collects the firmware measurement during the measured boot and sends the signed measurement to the model and data provider during the key establishment. The model and data provider detect if the NPU runs an older or compromised firmware version. The attacker cannot boot an emulated NPU, as it does not have the private NPU key. Denial of service (DoS) is out of the scope of GuardAIn.

**Malicious model provider.** The motive of a malicious model provider is to steal data from the data provider. A malicious model provider can manipulate the AI operator code to execute unintended operations. For example, it can copy part of the data to the model output and retrieve it later. By encrypting the output with the data provider's key before remapping it, we ensure that only the data provider can decrypt it. GuardAIn does not prevent kernels from overwriting the model (relevant for training). However, overwriting models only leads to either a corrupted model or wrong results, which is not in the interest of the model provider. A bug in the model operator code allows an attacker to copy a part of the model binary or parameter to the data provider output. However, we consider such a case out of the scope of GuardAIn.

**Malicious data provider.** The memory invariants prevent the data provider from copying the model memory after its decryption, preventing the model from leaking. Once the attestation is complete, the model memory is locked. Locking the memory ensures the data provider cannot send malicious commands to the task buffer. GuardAIn prevents the declaration of model output that overlaps with already allocated memory. Therefore, it prevents the data provider from leaking model parameters or operators to the model output. A model stealing (MS) attack involves a malicious data provider stealing the model parameters or inferring the operator code. Another attack, known as a membership inference attack (MIA), enables the data provider to infer the training data by sending many queries to the model. In GuardAIn, MS and MIA are orthogonal problems and are out of scope. Independent of GuardAIn, the model provider can deploy additional measures in the operators to add noise to the input or reject inferences after a given number of queries. The model provider leverages GuardAIn's model and task attestation (refer to Sec. 5.3) to ensure such security measures are in place.

## 8. Implementation and Evaluation

### 8.1. GuardAIn Implementation

We implement GuardAIn into the Ascend 910A NPU's software stack, which involves the driver, firmware, runtime, and Ascend PyTorch adapter written in C++. The AES-GCM-128 AI CPU operator is based on the AArch64cryptolib [59] library that uses ARM's hardware cryptographic intrinsic for efficient execution. For training and inference of LLM models, we use both the Ascend native execution (ACL runtime) and the Huawei PyTorch adapter. We introduce $\sim$2 KLoC to implement GuardAIn and add support to the AI stack. We use the squad_v2 dataset [60] for LLM inference, cipfer10 [61] for training image models and tiny Shakespeare [62] for training LLMs.

**Model provider.** We emulate the model provider by implementing it as a TCP server that serves authenticated and encrypted compiled models using a shared secret previously negotiated with the NPU. The model provider appends the MAC of the parameters and operator instructions after each encrypted blob, increasing the size of each operator binary by 16 bytes.

**Host runtime.** We modify the NPU runtime (CANN) [63] to implement the model and task attestation (cf. Sec. 5.3). During the `loadModel` API call, we extract the tasks and their associated `PC_START` attribute, which points to the operator binary's memory location. The modified runtime expects an additional signed binary sequence from the model file, establishing the model's ground truth. The modified runtime communicates with the data provider over the TCP socket to retrieve the signed `PC_START` sequence.

**NPU Task scheduler (TS).** The modified NPU TS firmware enforces the memory invariant and the attestation. The TS firmware records all the tasks submitted since the completion of the last execution. Upon reception of the `executeModel` command, the TS firmware computes the signature of the `PC_START` sequence using $\mathcal{K}_D$

(cf. Sec. 8.1). It then verifies that the signature matches the one generated by the data provider and sent with the `executeModel` command. If all the steps above are successful, the TS firmware relays the `executeModel` command to the TS hardware.

**AI CPU operators.** We use the CANN [63] SDK's AI-CPU operator development environment [64] to implement the custom AI CPU operator. The custom operators are responsible for the AES-GCM decryption and encryption (cf. Sec. 5.1) operations, as well as carrying out model and task attestation (cf. Sec. 5.3). The AI CPU operators are executed whenever they are called from the compiled model graph. They can also be directly called in C++ or from PyTorch. All the AI CPU operators are implemented as internal operators, i.e., the compiled binary of the operator remains inside the trusted NPU firmware. Therefore, a user cannot modify the operator code.

**Memory lock.** We use the `dma_map_page` and `dma_unmap_page` functions to implement the NPU memory invariants (Sec. 5.2). The host only accesses mapped HBM regions on the device via DMA using shared virtual memory (SVM). Removing the mapping of the corresponding DMA addresses prevents unauthorized host access. To ensure the unmapping happens before the device starts verifying the tasks, we implement a memory lock in the NPU TS, responsible for scheduling tasks on the AI CPUs and AI Cores. The NPU TS sends a synchronized message (implemented over shared memory, non-accessible to the host) to the memory manager driver running on the NPU control CPU, which then uses `dma_unmap_page` to unmap the entire mapped region. Once the unmapping is complete, the NPU TS starts with the model and task attestation (Sec. 5.3). If the unmapping fails, the device aborts the execution. Upon encrypting the output and resetting the input, the AES operator informs the NPU control CPU, allowing the respective regions to be remapped.

## 8.2. GuardAIn Evaluations

In the following sections, we present the evaluation of GuardAIn. We first discuss microbenchmarks, where we evaluate smaller building blocks of GuardAIn, then we will show and discuss the end-to-end performance and overhead of LLMs. We evaluate GuardAIn on four different LLMs with different parameter sizes: A small one (for modern standards), GPT-Neo-125M, and three larger state-of-the-art ones, Llama2 (7 and 13 B) and Llama3 (8 B).

**8.2.1. Micro benchmarks.** The AES-GCM AI-CPU operator decrypts and verifies the model during setup and the data during inference, affecting both the setup and inference time. Fig. 10 shows the parallel AES-GCM-128 performance on all four AI-CPU cores of the Ascend 910A NPU. We observe a maximum throughput of 6.1 GB/s when the data size matches the shared L2 cache size of 1KB (i.e., 256B for each core). For data sizes exceeding 1K, the cores experience cache contention and converge to single-core performance,

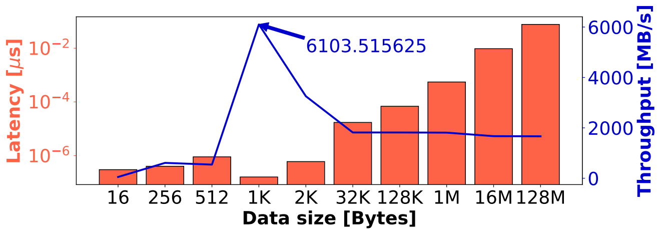


Figure 10. Parallelized AES-GCM-128 operator latency and throughput on the Ascend 910A AI-CPU cores on different data chunk sizes.

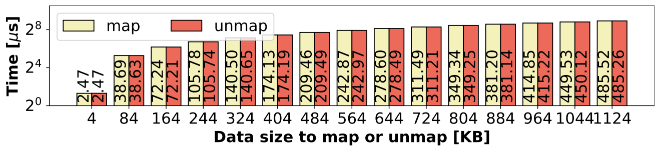


Figure 11. Ascend 910A's Map/unmap latency from host.

around 1.6 GB/s. To maximize throughput, the operator is fed a 1KB fixed chunk size (with interleaved DMA transfer).

Fig. 11 shows the latency of the map (`dma_map_pages`) and unmap (`dma_unmap_pages`) calls from the NPU driver to lock or unlock DMA memory regions for the host. In our GuardAIn prototype, the map and unmap calls work at 4K page granularity, which takes 2.47 $\mu$s for both operations. Most unmap operations are done during the model setup to remove access to the entire DMA-mapped space. In the subsequent inference passes, GuardAIn only needs to unmap the input and output regions before decryption and remap the result after its encryption. Consequently, the number of pages that eventually need to be mapped or unmapped is relatively small. The mapping and remapping for Llama-2-7B (in fp16 precision) takes $\sim$7sec using 4K pages. However, forcing the NPU SVM to use huge pages (1G) helps to minimize the latency ($\sim$15$\mu$s/1G page). Note that the encryption and mapping/unmapping are *one-time* operations as the inference serving systems typically avoid cold-start [65] to reduce the latency of the first token generation.

**8.2.2. LLM Inference and Model Setup Evaluation.** We evaluate the GuardAIn setup and runtime overhead. The setup incurs a one-time cost when the model is loaded on the NPU (`loadModel` API) and prepared (lock 5.2, decryption 5.1, and attestation 5.3) for execution. The runtime overhead denotes the added latency for text generation during inference. Users typically only perceive the runtime overhead while interacting with LLM applications.

**Setup overhead.** In the following paragraph, we evaluate the setup overhead. Typically, the setup time reflects the size of the AI model, which is proportional to the model parameter count. Therefore, smaller models have significantly lower loading times than larger models, e.g., 0.25 seconds for GPT-Neo-125M vs 26 seconds for Llama-3 8B. In GPT-Neo, the setup time overhead is between 50-71%, as seen in Fig. 12. We observe that the GPT-neo with a 2K sequence length loads faster than the other sequence lengths. GPT-

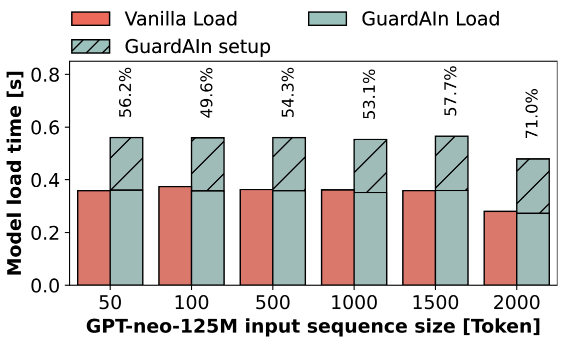


Figure 12. GPT-Neo-125 Load time overhead. The solid color boxes (Load) show `loadModel` execution time. The hatched boxes (setup) show the memory lock, model attestation, and decryption time.

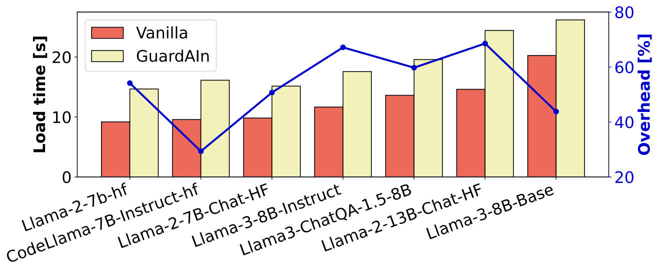


Figure 13. GUARDAIN load time overheads (%) in different Llama model variants with PyTorch.

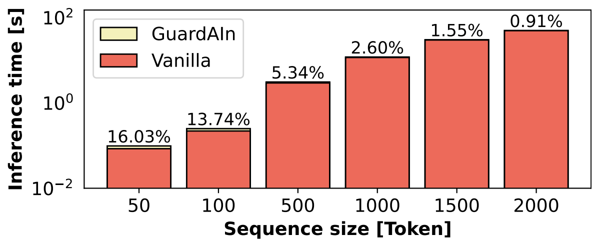


Figure 14. Inference time overhead (% on the bars) of GPT-Neo-125M with different input sequence sizes.

neo uses 2K as the default (maximum) sequence length. Any other sequence length causes the `loadModel` to add a padding layer to pad the input size to 2K, adding latency. The shorter load time of 2K sequence length makes the GUARDAIN overhead larger (71%). However, in absolute time, the overhead is consistent across all sequence lengths (0.17s) as the GUARDAIN overhead depends solely on the model size. A similar trend can also be seen in the Llama2 and Llama3 variants, depicted in Fig. 13. We also observe that Llama-3-8B-Base has the highest load time as it is encoded with `bfloat16` data type, unlike other models that are encoded with the Ascend-native `float16` data type. Ascend 910A NPU does not have native support for `bfloat16`. Hence, Pytorch converts the datatype from `bfloat16` to `float16`, which adds to the latency. In summary, GUARDAIN does not modify the `loadModel` latency; it only adds time for model locking, attestation, and decryption.

**Inference overhead.** Fig. 14, and Fig. 15 show the runtime overhead of GUARDAIN with GPT-Neo 125M. In GPT-Neo, the overhead is significantly higher than Llama's, as a smaller model (125M vs. 7/8/13 B) has significantly lower inference latency than a larger one (ms vs. seconds). For short input lengths (50 and 100), we observe 16.03% and 13.74% overhead. Larger sequences (e.g., actual chat queries with longer contexts) result in higher inference latency (quadratic to the sequence length). However, additional latency from GUARDAIN only increases linearly with the sequence size. This reduces overhead in larger sequence sizes, e.g., 0.91% in 2K sequence length.

GUARDAIN's overhead reduction in LLMs with a higher parameter count is apparent in models such as Llama-2, Llama-3, and CodeLlama, along with their variants (chat, instruct, and Q&A). These results are depicted in Fig. 15 for different context sizes. Note that we only evaluate up to a 2K context size for the 13B variant of Llama-2, as larger input sequence sizes in these two models caused out-of-memory errors. We observed a reduction of GUARDAIN overhead with larger input sequence sizes. This is expected as the inference latency with increasing sequence sizes is dominated by the model computation rather than the cryptographic operations. We generally observe less than 0.1% overhead in all Llama variants across all input sequence sizes. Note that we use a batch size of one in all of these experiments. With larger batch sizes, the overhead increases by a small fraction.

**Effect on accuracy.** As GUARDAIN does not modify the model structure or parameters, we did not observe any loss of accuracy compared to vanilla execution. We compare the GUARDAIN inference output with the vanilla output for cross-validation. We set the temperature to zero in all inference experiments to ensure deterministic output.

**8.2.3. LLM Training.** We evaluate GUARDAIN's performance overhead in model training. Unlike the inference evaluation, we train CNNs such as ResNet-50 and ResNet-152, and a small-size transformer model nanoGPT (162M) due to the immense cost of training large models such as Llama-2 7B (184K GPU hours [66]). Fig. 16 shows training overhead on ResNet 50 and 152 with different batch sizes over 200 epochs on the CIFER-10 image dataset and nanoGPT on the Tiny Shakespeare dataset. We observe overhead of $3.39\times10^{-4}$% in ResNet152 with batch size 256 and 0.16% in NanoGPT-162M with batch size 4.

**Effect on training loss.** During training with GUARDAIN, we did not observe any change in the learning rate or training loss compared to the vanilla training, as the model structures remain unchanged.

## 9. Related Work

Running sensitive computation inside a CPU-TEE (e.g., Intel SGX) [67] undermines the advantages of AI accelerators. Other approaches partially mitigate this shortcoming

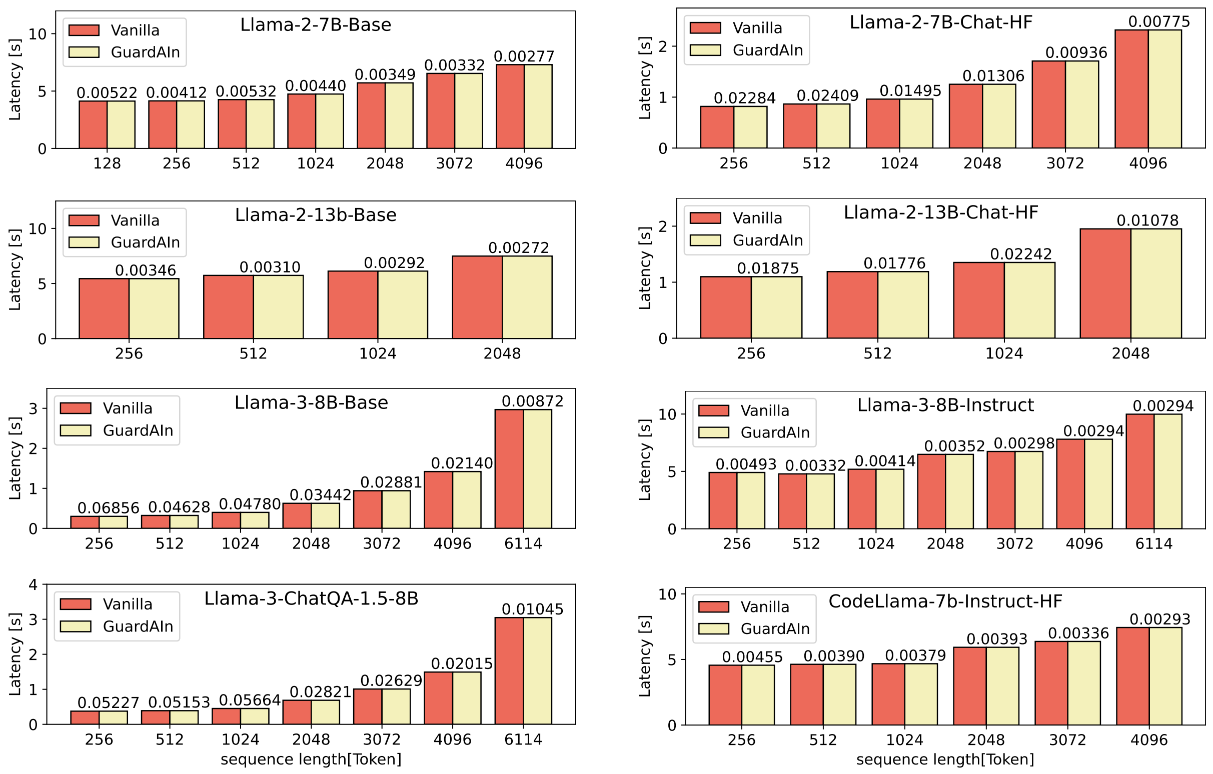


Figure 15. Single inference overhead in different input sequence sizes for vanilla and GUARDAIN on Llama2 and its variants. The value on the bar indicates the overhead of GUARDAIN over vanilla in percentage (%).

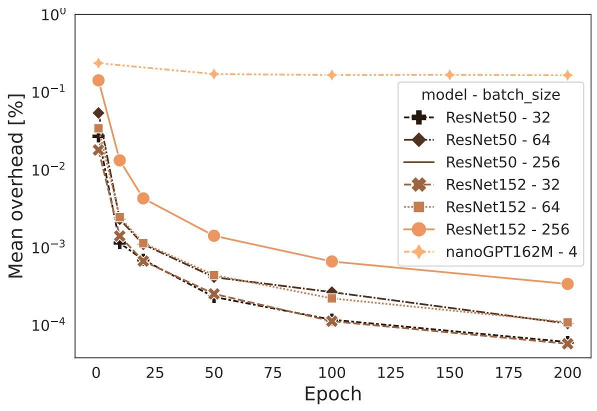


Figure 16. GUARDAIN training overhead on ResNet50, Resnet152 and nanoGPT162M on different batch sizes.

by only running selected parts of the workload inside the CPU-TEE while utilizing accelerators for more intensive tasks [68], [69], [70]. These approaches, though, still have a significant overhead and are vulnerable to privacy-stealing attacks [71]. In parallel (and as detailed in Sec. 3), there has been a long line of work aimed at directly extending the CPU-TEE or C-VMs to specific accelerators [15], [17], [22], [72], [73]. Graphcore [18] and SheF [20] move the trust entirely from the host to the device used to run the workload. Similarly, GuardNN [74] removes the trust from the host by redesigning an FPGA as an entirely new secure accelerator for ML tasks. On a larger scale, a few approaches [42], [75] try to extend the confidential computing paradigm to data-center architectures, allowing many devices to be split across users. Telekine [76] enables secure communication with cloud TEE-enabled GPUs [16]. Several existing works purely focus on improving memory isolation through improved capabilities [77], [78], or enhancing I/O isolation mechanisms for confidential computing [79], which often directly benefits TEE-devices architectures.

Other commercial solutions, such as Nvidia vGPU [80] or AMD MxGPU [81], use SR-IOV to virtualize GPUs and share them with multiple VMs. However, unlike GUARDAIN, such solutions require a trusted hypervisor and have a much larger TCB.

GUARDAIN does not provide any form of multi-tenancy (unlike Nvidia MIG [44]) as we point out in Sec. 3. Multi-tenancy often requires complex mechanisms to isolate memory between tenants. Current LLMs are often too large to fit multiple on a single device, which comes at the cost of an increased end-to-end token generation latency. Moreover, supporting multi-tenancy requires a trusted entity on the host, often a trusted hypervisor or a confidential VM (in the case of Nvidia MIG). GUARDAIN excludes the host CPU from the TCB.

Accelerator-enabled secure multi-party computation using secret sharing [82], [83] or homomorphic encryption [84], [85] incurs significant overhead in many real-world workloads [86], making them impractical for our setting.

Besides using a TPM for remote attestation, there have been a few examples of software-based attestation [87], [88], mainly for IoT devices. Upcoming PCIe features enable mechanisms to connect CPU-TEEs with DSA-TEEs. TDISP PCIe-6 [89] enables bounce-buffer-free shared encrypted memory between C-VM/CPU-TEEs devices. TDISP relies on PCI-IDE (Integrity and Data Encryption) [90], which requires a trusted motherboard/chipset/cloud provider. PCI-IDE on PCIe-5 uses AES-GCM for authenticated encryption of PCIe Transaction Layer Packets (TLPs) between the CPU and devices and between devices (P2P). TDISP reduces overhead by eliminating additional memory copy operations but necessitates additional hardware on the host and device sides, which GUARDAIN does not require.

The adoption of these PCIe extensions, such as TDX-Connect for Intel TDX, SEV-TIO for AMD SEV-SNP, Device Attach (DA) for Arm CCA, or IOPMP for RISC-V allows these TEE-enabled devices to benefit from secure direct access to the TEE memory on processors [91], [92], [93], [94]. Lastly, Table 1 comprehensively analyzes relevant existing works in multiple aspects compared to GUARDAIN.

## 10. Conclusion

We present GUARDAIN, a confidential computing solution on NPUs that secures models and data from an untrusted host and the cloud. GUARDAIN isolates NPU memory and enables encrypting models and data by leveraging the heterogeneous architecture of NPUs. The task and model attestation protects the integrity and confidentiality of the model and data. GUARDAIN implementation on a Huawei Ascend 910A NPU and evaluation of state-of-the-art generative AI workloads shows that GUARDAIN is practical, secure, and introduces minimal overhead.

## Acknowledgment

We thank Yuan Jinfeng for helping us navigate the Ascend driver, firmware, SVM, and documentation, and Xu Shifeng for helping us understand the inner workings of the Ascend NPU task scheduler.

## References

[1] OpenAI, "ChatGPT-OpenAI," [Accessed 12-06-2024].

[2] ——, "DALL-E-2 - OpenAI," [Accessed 12-06-2024].

[3] ——, "Sora - OpenAI," [Accessed 12-06-2024].

[4] Microsoft, "GitHub Copilot overview — code.visualstudio.com," [Accessed 12-06-2024].

[5] Google, "AI Infrastructure ML and DL Model Training — Google Cloud — cloud.google.com," [Accessed 12-06-2024].

[6] Microsoft, "Azure OpenAI Service – Advanced Language Models — Microsoft Azure — azure.microsoft.com," [Accessed 12-06-2024].

[7] Huawei, "Ascend AI Cloud Service — Huawei Cloud — huawei-cloud.com," [Accessed 12-06-2024].

[8] Alibaba, "Alibaba Cloud AI and Data Intelligence - Alibaba Cloud — alibabacloud.com," [Accessed 12-06-2024].

[9] C. Nast, "OpenAI's CEO Says the Age of Giant AI Models Is Already Over — wired.com," [Accessed 12-06-2024].

[10] S. Ray, "Samsung Bans ChatGPT Among Employees After Sensitive Code Leak — forbes.com," [Accessed 12-06-2024].

[11] Intel, "Intel Software Guard Extensions."

[12] AMD, "AMD SEV-SNP."

[13] ARM, "Learn the Architecture: TrustZone for AArch64," 2021.

[14] ——, "Arm Confidential Compute Architecture (ARM-CCA)."

[15] "NVIDIA Hopper Architecture In-Depth," 2022.

[16] S. Volos, K. Vaswani, and R. Bruno, "Graviton: Trusted execution environments on gpus," in *USENIX OSDI*, 2018.

[17] I. Jang, A. Tang, T. Kim, S. Sethumadhavan, and J. Huh, "Heterogeneous isolated execution for commodity gpus," in *ASPLOS*, 2019.

[18] K. Vaswani, S. Volos, C. Fournet, A. N. Diaz, K. Gordon, B. Vembu, S. Webster, D. Chisnall, S. Kulkarni, G. Cunningham *et al.*, "Confidential computing within an {AI} accelerator," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 501–518.

[19] S. Zeitouni, J. Vliegen, T. Frassetto, D. Koch, A.-R. Sadeghi, and N. Mentens, "Trusted configuration in cloud fpgas," in *IEEE FCCM*, 2021.

[20] M. Zhao, M. Gao, and C. Kozyrakis, "Shef: shielded enclaves for cloud fpgas," in *ACM ASPLOS*, 2022.

[21] H. Oh, K. Nam, S. Jeon, Y. Cho, and Y. Paek, "Meetgo: A trusted execution environment for remote applications on fpga," *IEEE Access*, vol. 9, pp. 51 313–51 324, 2021.

[22] S. Sridhara, A. Bertschi, B. Schlüter, M. Kuhne, F. Aliberti, and S. Shinde, "ACAI: Protecting accelerator execution with arm confidential computing architecture," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[23] C. Wang, F. Zhang, Y. Deng, K. Leach, J. Cao, Z. Ning, S. Yan, and Z. He, "Cage: Complementing arm cca with gpu extensions," in *Network and Distributed System Security (NDSS) Symposium*, 2024.

[24] E. Feng, D. Feng, D. Du, Y. Xia, and H. Chen, "snpu: Trusted execution environments on integrated npus," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2024.

[25] Y. Deng, C. Wang, S. Yu, S. Liu, Z. Ning, K. Leach, J. Li, S. Yan, Z. He, J. Cao *et al.*, "Strongbox: A gpu tee on arm endpoints," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[26] H. Park and F. X. Lin, "Safe and practical gpu computation in trustzone," in *Proceedings of the Eighteenth European Conference on Computer Systems*, 2023, pp. 505–520.

[27] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018.

[28] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[29] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How SGX amplifies the power of cache attacks," in *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2017.

[30] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: SGX cache attacks are practical," in *USENIX WOOT 17*, 2017.

[31] R. Paccagnella, L. Luo, and C. W. Fletcher, "Lord of the ring(s): Side channel attacks on the CPU on-chip ring interconnect are practical," in *USENIX Security*, 2021.

[32] N. Shrivastava and S. R. Sarangi, "Securator: A fast and secure neural processing unit," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023.

[33] L. Guo and F. X. Lin, "Minimum viable device drivers for arm trustzone," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. ACM, 2022.

[34] NVIDIA, "NVIDIA Grace Hopper Superchip — nvidia.com," [Accessed 24-08-2024].

[35] D. Abts, G. Kimmell, A. Ling, J. Kim, M. Boyd, A. Bitar, S. Parmar, I. Ahmed, R. DiCecco, D. Han *et al.*, "A software-defined tensor streaming multiprocessor for large-scale machine learning," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022.

[36] "Atlas 300T Training Card - Huawei Enterprise," [Accessed 17-06-2024].

[37] H. Liao, J. Tu, J. Xia, H. Liu, X. Zhou, H. Yuan, and Y. Hu, "Ascend: a scalable and unified architecture for ubiquitous deep neural network computing: Industry track paper," in *IEEE HPCA*, 2021.

[38] Huawei, "GitHub - Ascend/pytorch: Ascend PyTorch adapter (torch_npu). github.com," [Accessed 18-06-2024].

[39] "Private Cloud Compute: A new frontier for AI privacy in the cloud," 2024.

[40] Apple, "Operating system integrity - Apple Platform Security," [Accessed 04-11-2024].

[41] H. Mai, J. Zhao, H. Zheng, Y. Zhao, Z. Liu, M. Gao, C. Wang, H. Cui, X. Feng, and C. Kozyrakis, "Honeycomb: Secure and Efficient GPU Executions via Static Validation," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023.

[42] J. Zhu, R. Hou, X. Wang, W. Wang, J. Cao, B. Zhao, Z. Wang, Y. Zhang, J. Ying, L. Zhang *et al.*, "Enabling rack-scale confidential computing using heterogeneous trusted execution environment," in *IEEE S&P*, 2020.

[43] Microsoft, "Azure confidential Cloud - Protect Data In Use — Microsoft Azure."

[44] Nvidia, "NVIDIA Multi-Instance GPU User Guide :: Nvidia Tesla Documentation."

[45] V. A. Korthikanti, J. Casper, S. Lym, L. McAfee, M. Andersch, M. Shoeybi, and B. Catanzaro, "Reducing activation recomputation in large transformer models," *Proceedings of Machine Learning and Systems*, vol. 5, pp. 341–353, 2023.

[46] Microsoft, "Microsoft Copilot in Bing — bing.com," [Accessed 12-06-2024].

[47] perplexity, "Perplexity," [Accessed 12-06-2024].

[48] A. Holdings, "ARM system memory management unit architecture specification - SMMU architecture version 2.0," 2024.

[49] Apple, "Apple introduces M4 chip — apple.com," 2024, [Accessed 24-08-2024].

[50] AMD, "AMD Instinct™ MI300A Accelerators," [Accessed 20-03-2025].

[51] PyToech, "How Computational Graphs are Constructed in PyTorch," [Accessed 20-03-2025].

[52] Nvidia, "Advanced API Performance: Command Buffers," 2021, [Accessed 15-07-2024].

[53] Google, "An in-depth look at Google's first Tensor Processing Unit (TPU)," 2017, [Accessed 15-07-2024].

[54] F. Jentzsch, Y. Umuroglu, A. Pappalardo, M. Blott, and M. Platzner, "Radioml meets finn: Enabling future rf applications with fpga streaming architectures," *IEEE Micro*, vol. 42, no. 6, pp. 125–133, 2022.

[55] AMD, "FINN: Dataflow compiler for QNN inference on FPGAs," [Accessed 20-03-2025].

[56] Intel, "OpenVINO™ toolkit: An open source AI toolkit that makes it easier to write once, deploy anywhere." [Accessed 20-03-2025].

[57] AMD, "AMD Vitis™ AI Software," [Accessed 20-03-2025].

[58] Huawei, "Atlas 800 Training Server User Guide (Model 9000,Liquid Cooling) 21," [Accessed 12-04-2025].

[59] ARM, "AArch64cryptolib," 2023.

[60] P. Rajpurkar, R. Jia, and P. Liang, "Know what you don't know: Unanswerable questions for squad," *CoRR*, vol. abs/1806.03822, 2018. [Online]. Available: http://arxiv.org/abs/1806.03822

[61] A. Krizhevsky *et al.*, "Learning multiple layers of features from tiny images," 2009.

[62] yusuketomoto, "tinyshakespeare," [Accessed 12-11-2024].

[63] Huawei, "CANN - Ascend Community," [Accessed 15-07-2024].

[64] ——, "Operator development,TBE&AI CPU Operator Development,API Reference,AI CPU API,Overview," [Accessed 15-07-2024].

[65] W. S. Yunfen Bai and J.-T. Hung, "How To Reduce Cold Start Times For LLM Inference," [Accessed 13-03-2025].

[66] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[67] T. Lee, Z. Lin, S. Pushp, C. Li, Y. Liu, Y. Lee, F. Xu, C. Xu, L. Zhang, and J. Song, "Occlumency: Privacy-preserving remote deep-learning inference using SGX," in *The 25th Annual International Conference on Mobile Computing and Networking*, 2019.

[68] F. Mo, A. S. Shamsabadi, K. Katevas, S. Demetriou, I. Leontiadis, A. Cavallaro, and H. Haddadi, "Darknetz: towards model privacy at the edge using trusted execution environments," in *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, 2020.

[69] F. Tramer and D. Boneh, "Slalom: Fast, verifiable and private execution of neural networks in trusted hardware," in *International Conference on Learning Representations*, 2018.

[70] H. Hashemi, Y. Wang, and M. Annavaram, "Darknight: An accelerated framework for privacy and integrity preserving deep learning using trusted hardware," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021.

[71] Z. Zhang, C. Gong, Y. Cai, Y. Yuan, B. Liu, D. Li, Y. Guo, and X. Chen, "No privacy left outside: On the (in-) security of tee-shielded dnn partition for on-device ml," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 3327–3345.

[72] J. Jiang, J. Qi, T. Shen, X. Chen, , S. Zhao, S. Wang, L. Chen, N. Zhang, X. Luo, and H. Cui, "Cronus: Fault-isolated, secure and high-performance heterogeneous computing for trusted execution environments," in *ACM/IEEE Micro*, 2022.

[73] W. Ren, W. Kozlowski, S. Koteshwara, M. Ye, H. Franke, and D. Chen, "Accshield: a new trusted execution environment with machine-learning accelerators," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023.

[74] W. Hua, M. Umar, Z. Zhang, and G. E. Suh, "Guardnn: secure accelerator architecture for privacy-preserving deep learning," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022.

[75] A. Dhar, S. Sridhara, S. Shinde, S. Capkun, and R. Andri, "Confidential Computing with Heterogeneous Devices at Cloud-Scale," in *Annual Computer Security Applications Conference (ACSAC)*. Applied Computer Security Associates (ACSA), 2024.

[76] T. Hunt, Z. Jia, V. Miller, A. Szekely, Y. Hu, C. J. Rossbach, and E. Witchel, “Telekine: Secure computing with cloud GPUs,” in *NSDI*, 2020.

[77] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, “The cheri capability model: Revisiting risc in an age of risk,” *ACM SIGARCH Computer Architecture News*, 2014.

[78] J. Z. Yu, C. Watt, A. Badole, T. E. Carlson, and P. Saxena, “Capstone: a capability-based foundation for trustless secure memory access,” in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 787–804.

[79] E. Feng, D. Feng, D. Du, Y. Xia, W. Zheng, S. Zhao, and H. Chen, “siopmp: Scalable and efficient i/o protection for tees,” in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024.

[80] NVIDIA, “Unlock Next Level Performance With NVIDIA Virtual GPUs,” [Accessed 13-03-2025].

[81] AMD, “MxGPU-Virtualization,” [Accessed 13-03-2025].

[82] S. Tan, B. Knott, Y. Tian, and D. J. Wu, “Cryptgpu: Fast privacy-preserving machine learning on the gpu,” in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021.

[83] N. Kumar, M. Rathee, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, “Cryptflow: Secure tensorflow inference,” in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020.

[84] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, “GAZELLE: A low latency framework for secure neural network inference,” in *27th USENIX security symposium (USENIX security 18)*, 2018.

[85] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, “Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy,” in *International conference on machine learning*. PMLR, 2016.

[86] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa, “Delphi: A cryptographic inference system for neural networks,” in *Proceedings of the 2020 Workshop on Privacy-Preserving Machine Learning in Practice*, 2020.

[87] A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla, “Swatt: Software-based attestation for embedded devices,” in *IEEE S&P*, 2004.

[88] A. Ibrahim, A.-R. Sadeghi, G. Tsudik, and S. Zeitouni, “Darpa: Device attestation resilient to physical attacks,” in *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, 2016, pp. 171–182.

[89] PCI-SIG, “PCI Express 6.0 Specification.”

[90] ——, “Integrity and Data Encryption (IDE) ECN Deep Dive,” accessed 2023-05-04.

[91] Intel, “Intel TDX Connect Architecture Specification,” 2023.

[92] AMD, “AMD SEV-TIO: Trusted I/O for Secure Encrypted Virtualization,” March 2023.

[93] A. Holdings, “Introducing Arm Confidential Compute Architecture guide Version 3.0,” 2023.

[94] sifive, “RISC-V Security Architecture Introduction,” 2019.

# Appendix A. Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

## A.1. Summary

This paper presents GuardAIn, a confidential computing architecture for NPUs for the purpose of LLM inference. This solution is agnostic to the host CPU TEE and allows mutual distrust not only between the model/data provider and CSP but also between the model and data providers themselves. The paper demonstrates the viability of the solution through a prototype implementation on Huawei’s Ascend 910A NPU.

## A.2. Scientific Contributions

1) Provides a valuable step forward in an established field.
2) Address a long-known issue.

## A.3. Reasons for Acceptance

1) This paper provides a valuable step forward in an established field of confidential computing for accelerator devices. The paper enables confidential computing with the NPU in a host-agnostic manner which reliance on a host based TEE. Most previous work in this domain has focused on CPU-TEEs and GPU-TEEs. Typically, most GPUs rely on NICs for scale-up to allow computation using very large models. Instead, using NPUs for LLM inference with confidential computing capabilities is certainly an interesting direction of research.
2) It also addresses a long-known issue of enabling mutual distrust between cloud providers, model owners and data providers when running interference workloads in the cloud. Several previous works have considered this problem but by enabling this trust model in a CPU agnostic manner, this paper explores a different point in the design space.

## A.4. Noteworthy Concerns

1) GuardAIn does not enable multi-tenancy, which is a significant drawback for cloud settings.
2) The source code and implementation for GuardAIn will not be made available publicly. This might make reproducing this solution a challenge. However, this was considered an acceptable trade-off to allow industry submissions to S&P where it is not always feasible to make the solution available publicly.
3) This solution has been demonstrated only in the context of relatively small LLMs (Llama-7/13B).

4) GUARDAIN includes extensive firmware changes to enable the desired security properties in a closed-source system. Replicating this framework on other types of NPUs (e.g., those based on FPGAs) may entail hardware changes. So, the paper would benefit from adding a section about extensibility to other types of NPUs.

## Appendix B. Response to the Meta-Review

1) In Sec. 3 and Sec. 6, we discuss the trade-off of single-tenant vs multi-tenant, specifically model size, inference latency, complexity, and dependency on a trusted execution on the CPU. Given these tradeoffs, we explicitly design GUARDAIN to be a single-tenant solution, and we consider not supporting multi-tenancy as a drawback of GUARDAIN.
2) [No response.]
3) This is true that GUARDAIN is demonstrated on relatively small LLMs (7/13B). However, we would like to point out that running a larger model does not change GUARDAIN's design decisions and is primarily an engineering effort. In larger models, GUARDAIN's runtime overhead will reduce as larger models will spend more time in AI-related computation.
4) In sec. 6 (Generalization to other AI Accelerators), we provide some intuition on how GUARDAIN can be extended to other AI accelerators. However, this may require hardware changes; therefore, a complete evaluation of these devices is out of the scope of the current paper.